unray Documentation

Release 0.1.0.dev

Martin Sandve Alnæs

Apr 10, 2018

Installation and usage

Version: 0.1.0.dev

Jupyter Widget for Volume Rendering of Unstructured Tetrahedral Mesh Data

CHAPTER 1

Quickstart

To get started with unray, install with pip:

```
pip install unray
```

CHAPTER 2

Contents

2.1 Installation

The simplest way to install unray is via pip:

```
pip install unray
```

or via conda:

```
conda install unray
```

If you installed via pip, and notebook version < 5.3, you will also have to install / configure the front-end extension as well. If you are using classic notebook (as opposed to Jupyterlab), run:

```
jupyter nbextension install [--sys-prefix / --user / --system] --py unray

jupyter nbextension enable [--sys-prefix / --user / --system] --py unray
```

with the appropriate flag. If you are using Jupyterlab, install the extension with:

```
jupyter labextension install unray
```

2.2 Introduction

2.3 Examples

This section contains several examples generated from Jupyter notebooks. The widgets have been embedded into the page for demonstrative pruposes.

2.3.1 Demonstration

Importing dependencies

```
In [1]: # We'll use numpy for representing raw arrays
        import numpy as np

        # ipywidgets is the framework for handling GUI elements
        # and communication between the python and browser context
        import ipywidgets as widgets
        from ipywidgets import VBox, HBox

        # pythreejs provides a scenegraph by mirroring
        # three.js objects as ipywidgets
        import pythreejs as three

        # Finally the unray library
        import unray as ur
```

Some helper functions

Setting up a scene with pythreejs involves some boilerplate code. These helper functions simplify a bit. They will be replaced with utilities from the threeplot library when it becomes available.

```
In [2]: def setup_renderer(group, *,
                           scale=1.0, camera_direction=(1, 1, 1), light_direction=(0, 1, 1),
                           width=800, height=600, background='#eeeeee'):
            "Helper function to setup a basic pythreejs renderer and scene, adding given group to it
            camera_position = tuple(map(lambda x: x*scale, camera_direction))
            light_position = tuple(map(lambda x: x*scale, light_direction))
            camera = three.PerspectiveCamera(
                position=camera_position,
                aspect=width/height
            )
            key_light = three.DirectionalLight(position=light_position)
            ambient = three.AmbientLight(intensity=0.5)
            scene = three.Scene(
                children=[key_light, ambient, camera, group],
                background=background)
            controls = three.OrbitControls(camera)
            renderer = three.Renderer(scene, camera, [controls],
                                      width=width, height=height)
            return renderer

        def display_plots(*plots, **kwargs):
            "Display all plots in a single renderer. Returns renderer."
            group = three.Group()
            for plot in plots:
                group.add(plot)
            renderer = setup_renderer(group, **kwargs)
            return renderer
```

Setup some data for testing

All unray plots need a mesh in the form of vertex coordinates in a M x 3 points array and vertex indices for each tetrahedron in a N x 4 cells array. Data for continuous piecewise linear functions is passed as length M arrays.

(Discontinuous DP1 or P0 functions are not yet supported.)

```
In [3]: def single_tetrahedron():
            cells = np.zeros((1, 4), dtype="int32")
            coordinates = np.zeros((4, 3), dtype="float32")
            cells[0, :] = [0, 1, 2, 3]
            coordinates[0, :] = [0, 0, 0]
            coordinates[1, :] = [1, 0, 0]
            coordinates[2, :] = [0, 1, 0]
            coordinates[3, :] = [0, 0, 1]
            values = np.zeros(4, dtype="float32")
            values[:] = [1, 3, 2, -1]
            return cells, coordinates, values

        def load_data(filename):
            mesh_data = np.load(filename)
            cells_array = mesh_data["cells"].astype(np.int32)
            points_array = mesh_data["points"].astype(np.float32)
            return cells_array, points_array

        def compute_example_function(points_array):
            # Coordinates of all vertices in mesh
            x = list(points_array.T)  # x[2] = z coordinate array for all vertices

            # Model center 3d vector
            center = list(map(lambda x: x.mean(), x))

            # Coordinates with origo shifted to center of model
            xm = list(map(lambda x, mp: x - mp, x, center))

            # Distance from model center
            xd = np.sqrt(sum(map(lambda x: x**2, xm)))
            radius = xd.max()

            # A wave pattern from the center of the model
            freq = 4
            func_wave = 2.0 + np.sin((freq * 2 * np.pi / radius) * xd)

            return func_wave

        # Example data
        #filename = None
        filename = "../data/heart.npz"
        #filename = "../data/brain.npz"
        #filename = "../data/aneurysm.npz"

        if filename:
            cells_array, points_array = load_data(filename)
            function_array = compute_example_function(points_array)
        else:
            # Single tetrahedron example
            cells_array, points_array, function_array = single_tetrahedron()
```

Efficiency and memory usage

The unray API can in many places take pure numpy arrays with data. To save memory and network traffic (copying between the python and browser context), it is highly recommended to create data objects wrapping the numpy arrays before setting up the plot objects. This will allow sharing data between plot objects on the browser side and

simultaneous updating of fields across multiple plots.

```
In [4]: # Define a reusable Mesh object from the arrays with cell and point data
        mesh = ur.Mesh(cells=cells_array, points=points_array)

        # Define a reusable Field object over the mesh with values in mesh vertices
        field = ur.Field(mesh=mesh, values=function_array)

        # Mesh diameter, used for positioning below
        scale = max(*[points_array[:,i].max() - points_array[:,i].min() for i in (0,1,2)])
```

Surface plot

The surface plot draws the facets of the mesh as solid opaque surfaces. It can display just the mesh, be configured with wireframe parameters, or show a scalar field mapped to colors on its surface. All plot objects support restriction to cells.

```
In [5]: # Display just the mesh
        plot = ur.SurfacePlot(mesh=mesh)
        display_plots(plot, scale=scale)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

```
In [6]: # Enable wireframe
        wp = ur.WireframeParams(enable=True)
        plot = ur.SurfacePlot(mesh=mesh, wireframe=wp)
        display_plots(plot, scale=scale)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

```
In [7]: # Configure colors of surface and wireframe
        wp = ur.WireframeParams(enable=True, color="#0000ff")
        color = ur.ColorConstant(color="#ff8888")
        plot = ur.SurfacePlot(mesh=mesh, color=color, wireframe=wp)
        VBox([display_plots(plot, scale=scale), plot.dashboard()])
```

```
VBox(children=(Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.04519939422
```

```
In [8]: # Map a scalar field to the default colormap
        color = ur.ColorField(field=field)
        plot = ur.SurfacePlot(mesh=mesh, color=color)
        display_plots(plot, scale=scale)
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

```
In [9]: # Specify color lookup table as an array of rgb triplets
        lut = ur.ArrayColorMap(values=[[0.2, 0, 0], [1.0, 0, 0]])
        color = ur.ColorField(field=field, lut=lut)
        wp = ur.WireframeParams(enable=True, color="#00aaaa", opacity=0.1)
        plot = ur.SurfacePlot(mesh=mesh, color=color, wireframe=wp)
        display_plots(plot, scale=scale, background="white")
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

Xray

The xray is a simple direct volume rendering mode with pure absorption of background light at every point in the mesh. The total absorption along a view ray behind each pixel becomes the opacity of the mesh at that point. The image projected to screen is then simply the background image scaled by the transparency of the mesh (transparency = 1 - opacity). It works best with a bright background since it only subtracts from existing color.

```
In [10]: # Try with default density (may be very dark)
         plot = ur.XrayPlot(mesh=mesh)
         display_plots(plot, scale=scale, background="white")
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

```
In [11]: # Define a constant density value
         plot = ur.XrayPlot(mesh=mesh, density=ur.ScalarConstant(value=0.15))
         display_plots(plot, scale=scale, background="white")
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

Sum projection

The sum projection is a simple direct volume rendering mode with pure emission of light at every point in the mesh. The image projected to screen is then simply the integral of emitted light along a view ray behind each pixel. It works best with a dark background because it only adds color on top.

```
In [12]: # With a saturated constant color the result is flat
         color = ur.ColorConstant(color="#ff004d")
         plot = ur.SumPlot(mesh=mesh, color=color, exposure=0.0)

         # All plots can setup some widgets for their traits with plot.dashboard()
         widgets.VBox([
             plot.dashboard(),
             display_plots(plot, scale=scale, background="black")
         ])
         # Try adjusting the exposure slider!
```

```
VBox(children=(Accordion(children=(VBox(children=(FloatSlider(value=1.0, description='Intensity'), Co
```

```
In [13]: # For spatially varying fields, the sum projection can quickly become incomprehensible,
         # it's probably best to stick to a single-hue color map
         lut = ur.ArrayColorMap(values=[[1,0,0], [0,0,1]])
         color = ur.ColorField(field=field, lut=lut)
         plot = ur.SumPlot(mesh=mesh, color=color, exposure=-0.0)
         display_plots(plot, scale=scale, background="black")
```

```
Renderer(camera=PerspectiveCamera(aspect=1.3333333333333333, position=(10.045199394226074, 10.0451993
```

Integration with ipywidgets

Attributes of the data and plot widgets can be linked with sliders and other GUI elements from ipywidgets for some interactive control.

```
In [14]: # Configure some plot with scalar attributes
         density = ur.ScalarConstant(value=1.0)
         color = ur.ColorConstant(color="#ff00ff")
         plot1 = ur.XrayPlot(mesh=mesh, density=density)
         plot2 = ur.SumPlot(mesh=mesh, color=color)

         # Note that we catch the renderer widgets here instead of displaying directly
         renderer1 = display_plots(plot1, scale=scale, width=400, height=400, background="white")
         renderer2 = display_plots(plot2, scale=scale, width=400, height=400, background="black")

         # Setup some widgets
         density_slider = widgets.FloatSlider(value=density.value, min=0.0, max=2.0, description="Den
         extinction_slider = widgets.FloatSlider(value=plot1.extinction, min=0.0, max=3.0, descriptio
         exposure_slider = widgets.FloatSlider(value=plot2.exposure, min=-3.0, max=3.0, description="
         color_picker = widgets.ColorPicker(value=color.color, description="Color")
```

```
        # Link widgets to plot attributes
        widgets.jslink((density_slider, "value"), (plot1.density, "value"))
        widgets.jslink((extinction_slider, "value"), (plot1, "extinction"))
        widgets.jslink((color_picker, "value"), (plot2.color, "color"))
        widgets.jslink((exposure_slider, "value"), (plot2, "exposure"))

        # Group renderer with sliders for a single output
        # (this is not necessary, sliders and renderer
        # can also be in different cell outputs)
        widgets.HBox([
            widgets.VBox([density_slider, extinction_slider, renderer1]),
            widgets.VBox([color_picker, exposure_slider, renderer2])
        ])
```

```
HBox(children=(VBox(children=(FloatSlider(value=1.0, description='Density', max=2.0), FloatSlider(val
```

Next cell shows how using plot.dashboard() simplifies the process for common cases.

All plot types support restricting the drawing to a subset of the cells defined by an indicator field

```
In [15]: # Setup an indicator field and select a random subset
         # of the cells to display in the surface plot,
         # with the remainder rendered using xray
         num_cells = cells_array.shape[0]
         indicators_array = np.zeros(num_cells, dtype="int32")
         x0 = points_array[cells_array[:,0],0]  # x coordinate of first vertex in each cell
         x0min = x0.min()
         x0max = x0.max()
         x0rel = (x0 - x0min) / (x0max - x0min)
         threshold = x0min + 0.5 * (x0max - x0min)
         indicators_array[np.where(x0 > threshold)] = 1
         indicators = ur.IndicatorField(mesh=mesh, values=indicators_array)

         # Setup xray plot
         restrict1 = ur.ScalarIndicators(field=indicators, value=0)
         density = ur.ScalarConstant(value=1.0)
         plot1 = ur.XrayPlot(mesh=mesh, restrict=restrict1, density=density)

         # Setup surface plot
         restrict2 = ur.ScalarIndicators(field=indicators, value=1)
         color = ur.ColorConstant(color="#008888")
         wp = ur.WireframeParams(enable=True)
         plot2 = ur.SurfacePlot(mesh=mesh, restrict=restrict2, color=color, wireframe=wp)

         # This time we add the plots to a single renderer
         renderer = display_plots(plot1, plot2, scale=scale, width=800, height=400, background="white

         # Using the plot.dashboard() function simplifies the
         # widget setup if you're happy with the defaults.
         # (The resulting widget setup is currently a bit crude)
         VBox([
             renderer,
             HBox([plot1.dashboard(), plot2.dashboard()])
         ])
```

```
VBox(children=(Renderer(camera=PerspectiveCamera(aspect=2.0, position=(10.045199394226074, 10.0451993
```

```
In [16]: # Trick to refresh plots until child events are handled correctly
```

```
plot1.visible = False
plot2.visible = False
plot1.visible = True
plot2.visible = True
```

That's all!

2.4 Developer install

To install a developer version of unray, you will first need to clone the repository:

```
git clone https://github.com/martinal/unray
cd unray
```

Next, install it with a develop install using pip:

```
pip install -e .
```

If you are planning on working on the JS/frontend code, you should also do a link installation of the extension:

```
jupyter nbextension install [--sys-prefix / --user / --system] --symlink --py unray

jupyter nbextension enable [--sys-prefix / --user / --system] --py unray
```

with the appropriate flag. Or, if you are using Jupyterlab:

```
jupyter labextension install .
```